Software carpentry

From theory to practice: Standard tools

Pietro Berkes, Enthought UK

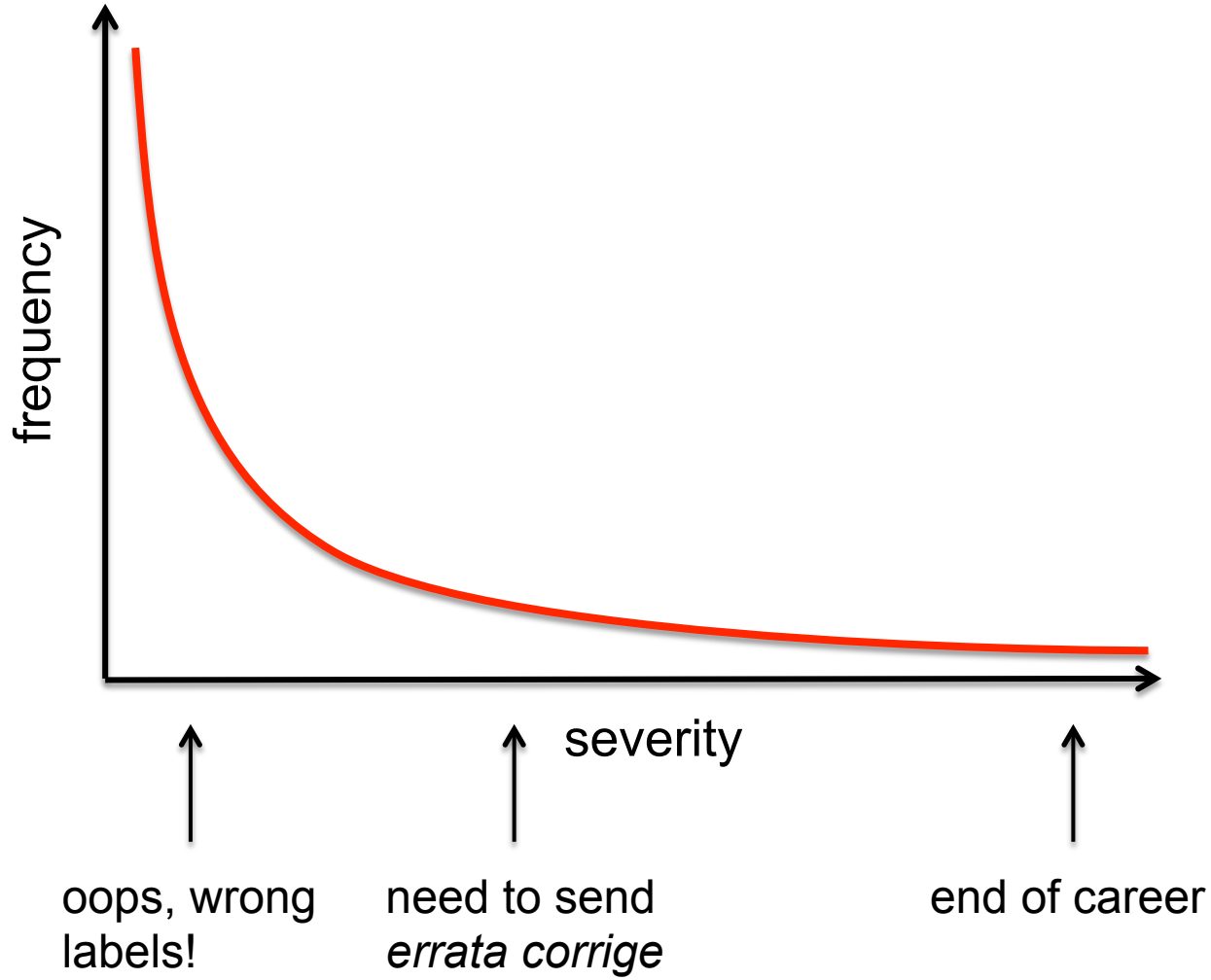
Modern programming practices and science

- Researchers and scientific software developers write software daily, but few have been trained to do so
- Good programming practices make a BIG difference
- We can learn a lot from the development methods developed for commercial and open source software in the past 10 years

Requirements for scientific programming

- Main requirement: scientific code must be **error free**
- Scientist time, not computer time is the bottleneck
  - Being able to explore many different models and statistical analyses is more important than a very fast single approach
- Reproducibility and re-usability:
  - Every scientific result should be independently reproduced at least internally before publication (DFG, 1999)
  - Increasing pressure for making the source code used in publications available online (especially for theoretical papers)
  - No need for somebody else to re-implement your algorithm

Effect of software errors in science



Software bugs in research are a serious business

Science, Dec 2006: 5 high-profile retractions (3x Science, PNAS, J. Mol. Biol.) because "an in-house data reduction program introduced a change in sign for anomalous differences"

SCIENTIFIC PUBLISHING

A Scientist's Nightmare: Software Problem Leads to Five Retractions

Until recently, Geoffrey Chang's career was on a trajectory most young scientists only dream about. In 1999, at the age of 28, the protein crystallographer landed a faculty position at the prestigious Scripps Research Institute in San Diego, California. The next year, in a cer-

2001 *Science* paper, which described the structure of a protein called MsbA, isolated from the bacterium *Escherichia coli*. MsbA belongs to a huge and ancient family of molecules that use energy from adenosine triphosphate to transport molecules across cell membranes. These

LETTERS

edited by Etta Kavanagh

Retraction

WE WISH TO RETRACT OUR RESEARCH ARTICLE "STRUCTURE OF MsbA from *E. coli*: A homolog of the multidrug resistance ATP binding cassette (ABC) transporters" and both of our Reports "Structure of the ABC transporter MsbA in complex with ADP•vanadate and lipopolysaccharide" and "X-ray structure of the EmrE multidrug transporter in complex with a substrate" (*1*–*3*).

The recently reported structure of Sav1866 (*4*) indicated that our MsbA structures (*1*, *2*, *5*) were incorrect in both the hand of the structure and the topology. Thus, our biological interpretations based on these inverted models for MsbA are invalid.

An in-house data reduction program introduced a change in sign for anomalous differences. This program, which was not part of a conventional data processing package, converted the anomalous pairs (I+ and I–) to (F– and F+), thereby introducing a sign change. As the diffraction data collected for each set of MsbA crystals and for the EmrE crystals were processed with the same program, the structures reported in (*1*–*3*, *5*, *6*) had the wrong hand.

PLoS Comp Bio, July 2007: retraction because "As a result of a bug in the Perl script used to compare estimated trees with true trees, the clade confidence measures were sometimes associated with the incorrect clades."

Retraction: Measures of Clade Confidence Do Not Correlate with Accuracy of Phylogenetic Trees

Barry G. Hall, Stephen J. Salipante

In *PLoS Computational Biology*, volume 3, issue 3, doi:10.1371/journal.pcbi.0030051:

As a result of a bug in the Perl script used to compare estimated trees with true trees, the clade confidence measures were sometimes associated with the incorrect clades. The error was detected by the sharp eye of Professor Sarah P. Otto of the University of British Columbia. She noticed a discrepancy between the example tree in Figure 1B and the results reported for the gene *nuoK* in Table 1, and requested that she be sent all ten *nuoK* Bayesian trees. She painstakingly did a manual comparison of those trees with the true trees, concluded that for that dataset there was a strong correlation between clade confidence and the probability of a clade being true, and suggested the possibility of a bug in the Perl script. Dr. Otto put in considerable effort, and we want to acknowledge the generosity of that effort.

This includes the industry

LEGAL/REGULATORY | AUGUST 2, 2012, 9:07 AM | 357 Comments

Knight Capital Says Trading Glitch Cost It $440 Million

BY NATHANIEL POPPER

Brendan McDermid/Reuters

1 2 3 4 ➤

Errant trades from the Knight Capital Group began hitting the New York Stock Exchange almost as soon as the opening bell rang on Wednesday.

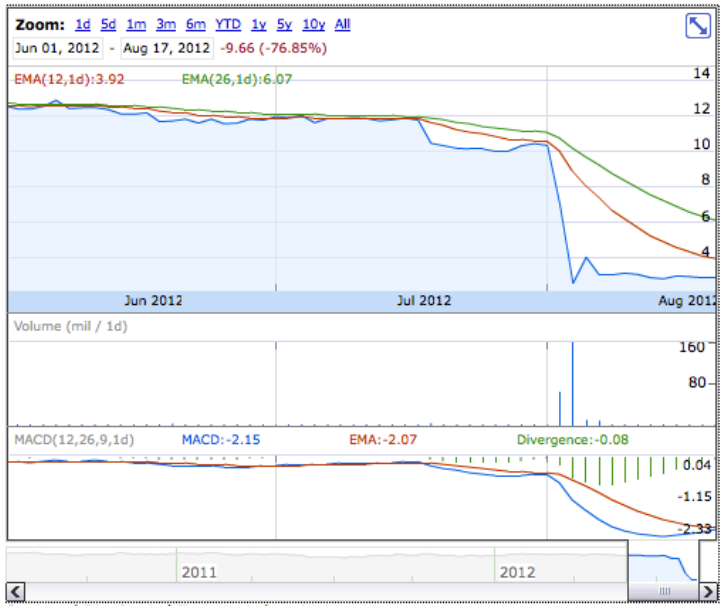
4:01 p.m. | Updated

$10 million a minute.

That's about how much the trading problem that set off turmoil on the stock market on Wednesday morning is already costing the trading firm.

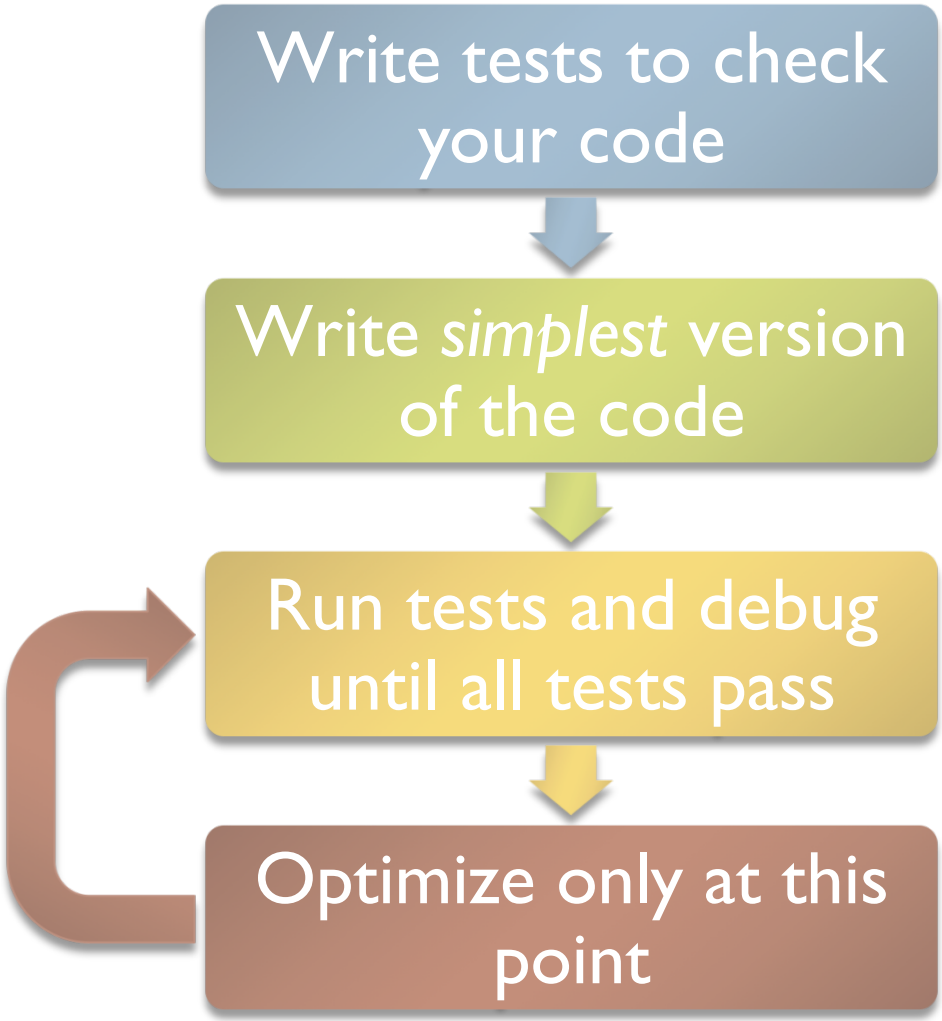
The Knight Capital Group announced on Thursday that it lost $440 million when it sold all the stocks it accidentally bought Wednesday morning because a computer glitch.

NYT, 2 August 2012



Source: Google Finance

The basic agile development cycle



`unittest`
`coverage.py`

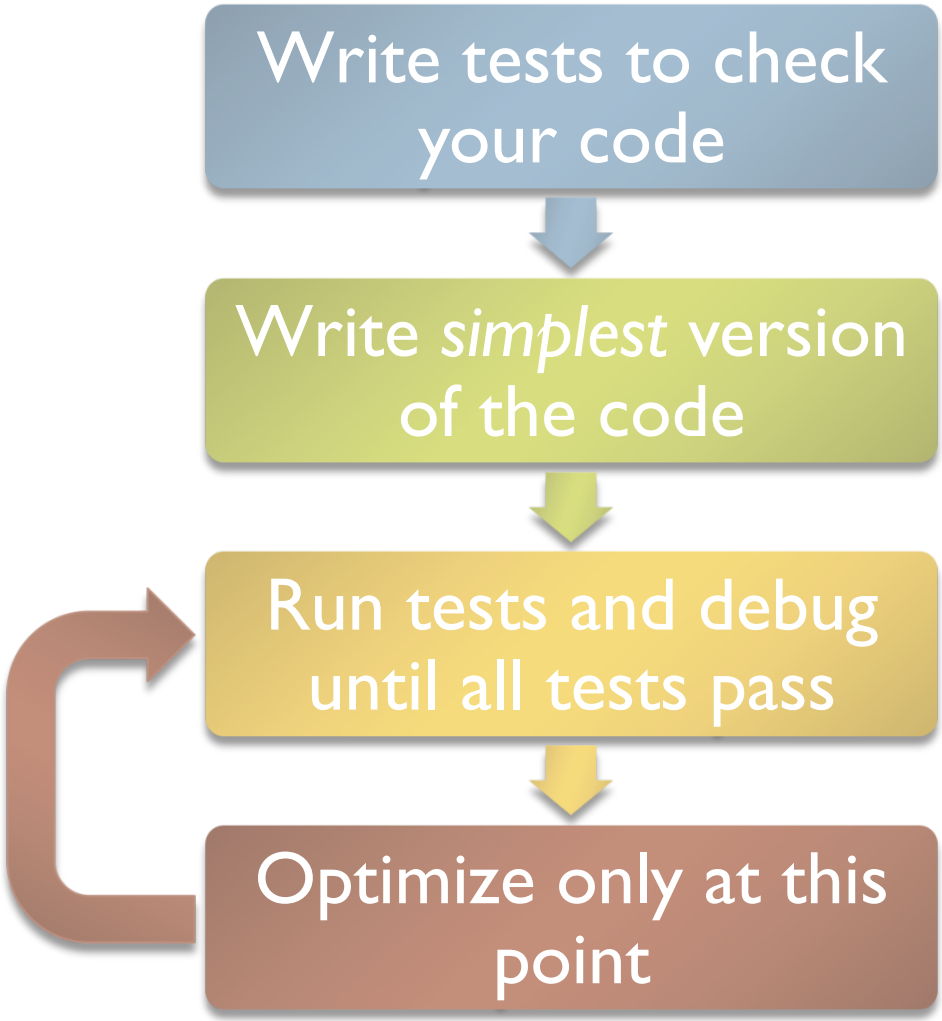
`pdb`

`cProfile`
`timeit`
`runSnake`

Python tools for agile programming

- There are many tools, based on command line or graphical interface
- I'll present:
  - Python standard "batteries included" tools
  - no graphical interface necessary
  - magic commands for ipython
- Alternatives and cheat sheets are on the wiki

The basic agile development cycle



unittest
coverage.py

`pdb`

`cProfile`
`timeit`
`runSnake`

Reminder: Testing in agile development

- Formal software testing has become one of the most important parts of modern software development
- Tests become part of the programming cycle and are automated:
  - Write test suite in parallel with your code
  - External software runs the tests and provides reports and statistics

```
test_choice (__main__.TestSequenceFunctions) ... ok
test_sample (__main__.TestSequenceFunctions) ... ok
test_shuffle (__main__.TestSequenceFunctions) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.110s
OK
```

Testing benefits

- Tests are the only way to trust your code
- Encourages better code and optimization: code can change, and consistency is assured by tests
- Faster development:
  - Bugs are always pinpointed
  - Avoids starting all over again when fixing one part of the code causes a bug somewhere else
- It might take you a while to get used to writing them, but it will pay off quite rapidly

Test-driven development (TDD)

- An influential testing philosophy: write tests *before* writing code
  - Choose what is the next feature you'd like to implement
  - Write a test for that feature
  - Write the simplest code that will make the test pass
- Forces you to think about the design of the interface to your code before writing it
- The result is code whose features can be tested individually, leading to maintainable, decoupled code

Testing with Python

- unittest:
  - Has been part of the Python standard library since v. 2.1
  - Interface a bit awkward (camelCase methods…), very basic functionality until...
  - Major improvement with 2.7, now at the level of other modern testing tools
  - Backward compatible, unittest2 back-port for earlier versions of Python
- Alternatives:
  - nosetests
  - py.test

Test suites in Python: `unittest`

- `unittest`: standard Python testing library

- Each test case is a subclass of `unittest.TestCase`
- Each test unit is a method of the class, whose name starts with '`test`'
- Each test unit checks **one** aspect of your code, and raises an exception if it does not work as expected

Anatomy of a TestCase

Create new file, `test_something.py`:

```
import unittest

class FirstTestCase(unittest.TestCase):

    def test_truisms(self):
        """All methods beginning with 'test' are executed"""
        self.assertTrue(True)
        self.assertFalse(False)

    def test_equality(self):
        """Docstrings are printed during executions
        of the tests in the Eclipse IDE"""
        self.assertEqual(1, 1)

if __name__ == '__main__':
    unittest.main()
```

Multiple TestCases

```
import unittest

class FirstTestCase(unittest.TestCase):

    def test_truisms(self):
        self.assertTrue(True)
        self.assertFalse(False)

class SecondTestCase(unittest.TestCase):

    def test_approximation(self):
        self.assertAlmostEqual(1.1, 1.15, 1)

if __name__ == '__main__':
    # execute all TestCases in the module
    unittest.main()
```

`TestCase.assertSomething`

- `TestCase` defines utility methods to check that some conditions are met, and raise an exception otherwise
- Check that statement is true/false:

```
assertTrue('Hi'.islower())          => fail
assertFalse('Hi'.islower())         => pass
```

- Check that two objects are equal:

```
assertEqual(2+1, 3)                 => pass
assertEqual([2]+[1], [2, 1])        => pass
assertNotEqual([2]+[1], [2, 1])     => fail
```

`assertEqual` can be used to compare numbers, lists, tuples, dicts, sets, frozensets, and unicode objects

TestCase.assertSomething

- Check that two numbers are equal up to a given precision:
  `assertAlmostEqual(x, y, places=7)`
- `places` is the number of decimal places to use:
  `assertAlmostEqual(1.121, 1.12, 2)` => pass
  `assertAlmostEqual(1.121, 1.12, 3)` => fail

Formula for almost-equality is

```
round(x - y, places) == 0.
```

and so

```
assertAlmostEqual(1.126, 1.12, 2)  => fail
```

TestCase.assertSomething

- One can also specify a maximum difference:

```
assertAlmostEqual(x, y, delta=0.)
```

E.g.:

```
assertAlmostEqual(1.125, 1.12, 0.06)  => pass
assertAlmostEqual(1.125, 1.12, 0.04)  => fail
```

- Can be used to compare any object that supports subtraction and comparison:

```
import datetime
delta = datetime.timedelta(seconds=10)
second_timestamp = datetime.datetime.now()

self.assertAlmostEqual(first_timestamp,
    second_timestamp, delta=delta)
```

TestCase.assertSomething

- Check that an exception is raised:

```
assertRaises(exception_class, function,
             arg1, arg2, kwarg1=None, kwarg2=None)
```

executes

```
function(arg1, arg2, kwarg1=None, kwarg2=None)
```

and passes if an exception of the appropriate class is raised

- For example:

```
assertRaises(IOError,
             file, 'inexistent', 'r')    => pass
```

Use the most specific exception class, or the test may pass because of collateral damage:

```
tc.assertRaises(IOError, file, 1, 'r')    => fail
tc.assertRaises(Exception, file, 1, 'r') => pass
```

TestCase.assertSomething

- The most convenient way to use assertRaises is as a context manager:

```
with self.assertRaises(SomeException):
    do_something()
```

For example:

```
with self.assertRaises(ValueError):
    int('XYZ')
```

passes, because

```
int('XYZ')
ValueError: invalid literal for int() with base 10: 'XYZ'
```

TestCase.assertSomething

- Many more "assert" methods:
  (complete list at http://docs.python.org/library/unittest.html)

```
assertGreater(a, b) / assertLess(a, b)
```

```
assertRegexpMatches(text, regexp)
```
verifies that regexp search matches text

```
assertIn(value, sequence)
```
assert membership in a container

```
assertIsNone(value)
```
verifies that value is None

```
assertIsInstance(obj, cls)
```
verifies that an object is an instance of a class

```
assertItemsEqual(actual, expected)
```
verifies equality of members, ignores order

```
assertDictContainsSubset(subset, full)
```
tests whether the entries in dictionary full are a superset of those in subset

TestCase.assertSomething

- Most of the `assert` methods accept an optional `msg` argument that overwrites the default one:

```
assertTrue('Hi'.islower(),
           'One of the letters is not lowercase')
```

- Most of the `assert` methods have a negated equivalent, e.g.:

```
assertIsNone
assertIsNotNone
```

Testing with numpy arrays

- When testing numerical algorithms, numpy arrays have to be compared elementwise:

```
class NumpyTestCase(unittest.TestCase):
    def test_equality(self):
        a = numpy.array([1, 2])
        b = numpy.array([1, 2])
        self.assertEqual(a, b)
```

```
E
======================================================================
ERROR: test_equality (__main__.NumpyTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "numpy_testing.py", line 8, in test_equality
self.assertEqual(a, b)
  File "/Library/Frameworks/Python.framework/Versions/6.1/lib/
python2.6/unittest.py", line 348, in failUnlessEqual
    if not first == second:
ValueError: The truth value of an array with more than one element is
ambiguous. Use a.any() or a.all()

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (errors=1)
```

Testing with numpy arrays

- `numpy.testing` defines appropriate function:

```
numpy.testing.assert_array_equal(x, y)
numpy.testing.assert_array_almost_equal(x, y,
                                        decimal=6)
```

- If you need to check more complex conditions:
  - `numpy.all(x)`: returns True if all elements of x are true

    `numpy.any(x)`: returns True is any of the elements of x is true

    `numpy.allclose(x, y, rtol=1e-05, atol=1e-08)`: returns True if two arrays are element-wise equal within a tolerance; rtol is relative difference, atol is absolute difference

  - combine with `logical_and, logical_or, logical_not`:

```
# test that all elements of x are between 0 and 1
assertTrue(all(logical_and(x > 0.0, x < 1.0))
```

How to run tests

- Option 1: unittest.main() will execute all tests in all TestCase classes in a file

```
if __name__ == '__main__':
    unittest.main()
```

- Option 2: Execute all tests in one file

```
python -m unittest [-v] test_module
```

- Option 3: Discover all tests in all subdirectories

```
python -m unittest discover
```

Basics of testing

- What to test, and how?
- At the beginning, testing feels weird:
  1) It's obvious that this code works (not TDD…)
  2) The tests are longer than the code
  3) The test code is a duplicate of the real code

- What does a good test looks like?
- What should I test?
- Anything specific to scientific code?

Basic structure of test

- A good test is divided in three parts:
  - **Given**: Put your system in the right state for testing
    - Create objects, initialize parameters, define constants…
    - Define the expected result of the test
  - **When**: The key actions of the test
    - Typically one or two lines of code
  - **Then**: Compare outcomes of the key actions with the expected ones
    - Set of *assertions* regarding the new state of your system

Test simple but general cases

- Start with simple, general case
  - Take a realistic scenario for your code, try to reduce it to a simple example
- Tests for 'lower' method of strings

```
class LowerTestCase(unittest.TestCase):

    def test_lower(self):
        # Given
        string = 'HeLlO wOrld'
        expected = 'hello world'

        # When
        output = string.lower()

        # Then
        self.assertEqual(output, expected)
```

Test special cases and boundary conditions

- Code often breaks in corner cases: empty lists, None, NaN, 0.0, lists with repeated elements, non-existing file, …
- This often involves making design decision: respond to corner case with special behavior, or raise meaningful exception?

```
def test_empty_string(self):
    # Given
    string = ''
    expected = ''

    # When
    output = string.lower()

    # Then
    self.assertEqual(output, expected)
```

- Other good corner cases for string.lower():
  - 'do-nothing case': `string = 'hi'`
  - symbols: `string = '123 (!'`

Common testing pattern

- Often these cases are collected in a single test:

```
def test_lower(self):
    # Given
    # Each test case is a tuple of (input, expected_result)
    test_cases = [('HeLlO wOrld', 'hello world'),
                  ('hi', 'hi'),
                  ('123 ([?', '123 ([?'),
                  ('', '')]

    for string, expected in test_cases:
        # When
        output = string.lower()
        # Then
        self.assertEqual(output, expected)
```

Fixtures

- Tests require an initial state or *test context* in which they are executed (the "Given" part), which needs to be initialized and possibly cleaned up.
- If multiple tests require the same context, this fixed context is known as a *fixture*.
- Examples of fixtures:
  - Creation of a data set at runtime
  - Loading data from a file or database
  - Creation of *mock* objects to simulate the interaction with complex objects

setUp and tearDown

```
import unittest

class FirstTestCase(unittest.TestCase):

    def setUp(self):
        """setUp is called before every test"""
        pass

    def tearDown(self):
        """tearDown is called at the end of every test"""
        pass

    @classmethod
    def setUpClass(cls):
        """Called once before all tests in this class."""
        pass

    @classmethod
    def tearDownClass(cls):
        """Called once after all tests in this class."""
        pass

    # ... all tests here ...
```

Numerical fuzzing

- Use deterministic test cases when possible
- In most numerical algorithm, this will cover only over-simplified situations; in some, it is impossible
- Fuzz testing: generate random input
  - Outside scientific programming it is mostly used to stress-test error handling, memory leaks, safety
  - For numerical algorithm, it is often used to make sure one covers general, realistic cases
  - The input may be random, but you still need to know what to expect
  - Make failures reproducible by saving or printing the random seed

Numerical fuzzing – example

```
class VarianceTestCase(unittest.TestCase):

    def setUp(self):
        self.seed = int(numpy.random.randint(2**31-1))
        numpy.random.seed(self.seed)
        print 'Random seed for the tests:', self.seed

    def test_var(self):
        N, D = 100000, 5

        # goal variances: [0.1 ,  0.45,  0.8 ,  1.15,  1.5]
        desired = numpy.linspace(0.1, 1.5, D)

        # test multiple times with random data
        for _ in range(20):
            # generate random, D-dimensional data
            x = numpy.random.randn(N, D) * numpy.sqrt(desired)
            variance = numpy.var(x, axis=0)
            numpy.testing.assert_array_almost_equal(variance, desired, 1)
```

Testing learning algorithms

- Learning algorithms can get stuck in local maxima, the solution for general cases might not be known (e.g., unsupervised learning)
- Turn your validation cases into tests
- Stability tests:
  - start from final solution; verify that the algorithm stays there
  - start from solution and add a small amount of noise to the parameters; verify that the algorithm converges back to the solution
- Generate data from the model with known parameters
  - E.g., linear regression: generate data as y = a*x + b + noise for random a, b, and x, then test that the algorithm is able to recover a and b

Other common cases

- Test general routines with specific ones
  - Example: test `polyomial_expansion(data, degree)` with `quadratic_expansion(data)`

- Test optimized routines with brute-force approaches
  - Example: test `z = outer(x, y)` with

```
M, N = x.shape[0], y.shape[0]
z = numpy.zeros((M, N))
for i in range(M):
    for j in range(N):
        z[i, j] = x[i] * y[j]
```

Example: eigenvector decomposition

- Consider the function `values, vectors = eigen(matrix)`
- Test with simple but general cases:
  - use full matrices for which you know the exact solution (from a table or computed by hand)
- Test general routine with specific ones:
  - use the analytical solution for 2x2 matrices
- Numerical fuzzing:
  - generate random eigenvalues, random eigenvector; construct the matrix; then check that the function returns the correct values
- Test with boundary cases:
  - test with diagonal matrix: is the algorithm stable?
  - test with a singular matrix: is the algorithm robust? Does it raise appropriate error when it fails?


DEMO

Code coverage

- It's easy to leave part of the code untested
  Classics:
  - Feature activated by keyword argument
  - Exception raised for invalid input
- Coverage tools mark the lines visited during execution
- Use together with test framework to make sure all your code is covered

coverage.py

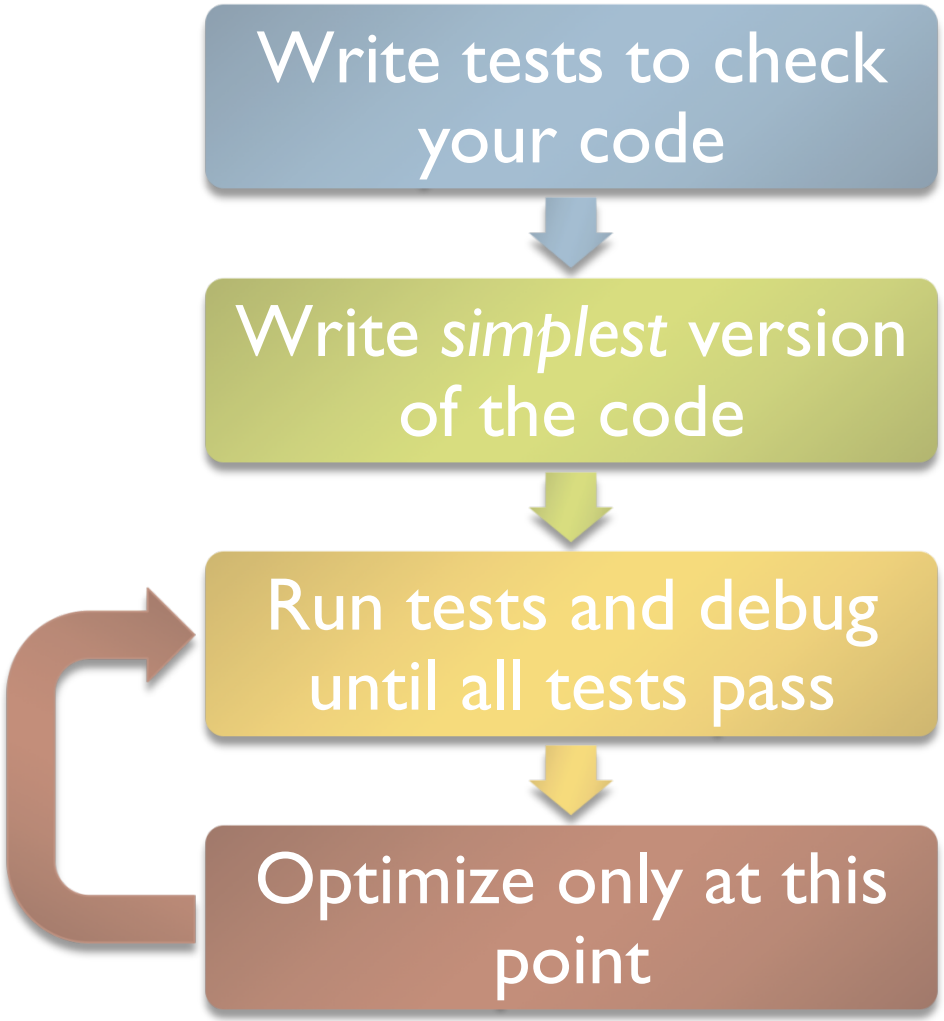
- Python script to perform code coverage
- Produces text and HTML reports
- Allows branch coverage analysis
- Not included in standard library, but quite standard


DEMO

Testing: Money back guarantee

- I guarantee that aggressive testing will improve your code, or you'll get the Python school fee back!
- Just remember, quality is not just testing:
  - Trying to improve the quality of software by doing more testing is like trying to lose weight by weighing yourself more often

The basic agile development cycle



`unittest`
`coverage.py`

`pdb`

`cProfile`
`timeit`
`runSnake`

Debugging

- The best way to debug is to avoid bugs
  - In TDD, you *anticipate* the bugs
- Your test cases should already exclude a big portion of the possible causes
- Core idea in debugging: you can stop the execution of your application at the bug, look at the state of the variables, and execute the code step by step
- Avoid littering your code with *print* statements

`pdb`, the Python debugger

- Command-line based debugger
- `pdb` opens an interactive shell, in which one can interact with the code
  - examine and change value of variables
  - execute code line by line
  - set up breakpoints
  - examine calls stack

Entering the debugger

- Enter debugger at the start of a file:

```
python -m pdb myscript.py
```

- Enter in a statement or function:

```
import pdb
# your code here
if __name__ == '__main__':
    pdb.runcall(function[, argument, ...])
    pdb.run(expression)
```

- Enter at a specific point in the code (alternative to `print`):

```
# some code here
# the debugger starts here
import pdb;pdb.set_trace()
# rest of the code
```

Entering the debugger from ipython

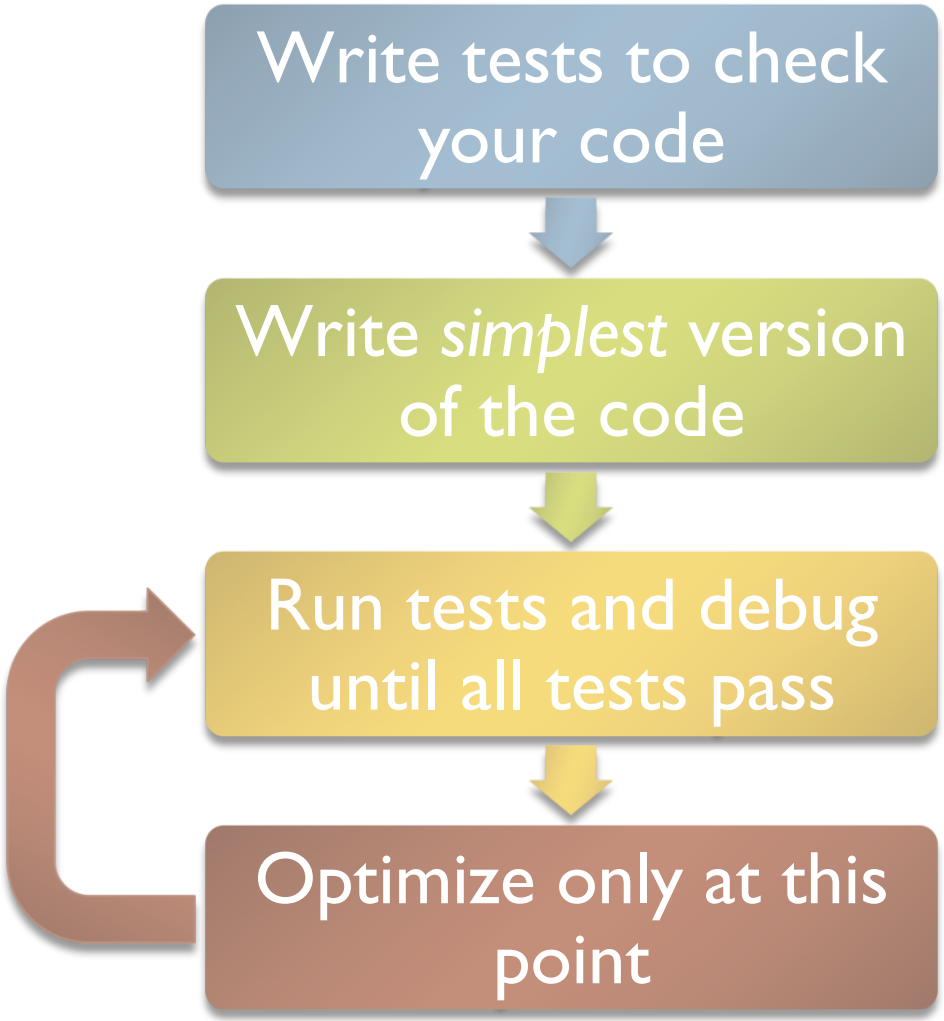
- From ipython:

  `%pdb` – preventive

  `%debug` – post-mortem


DEMO

The basic agile development cycle



`unittest`
`coverage.py`

`pdb`

`cProfile`
`timeit`
`runSnake`

Python code optimization

- Python is slower than C, but not prohibitively so
- In scientific applications, this difference is even less noticeable (`numpy`, `scipy`, ...)
- Don't rush into writing optimizations

How to optimize

- Usually, a small percentage of your code takes up most of the time
- Stop optimizing as soon as possible

1. Identify time-consuming parts of the code (use a profiler)
2. Only optimize those parts of the code
3. Keep running the tests to make sure that code is not broken

Optimization methods hierarchy

- **Warning: controversial content**
- In order of preference:
  - Vectorize code using numpy
  - Use a "magic optimization" tool, like numexpr, or scipy.weave
  - Spend some money on better hardware, optimized libraries (e.g., Intel's MKL)
  - Use Cython
  - Parallelize your code
  - Use GPU acceleration
- The problem might be about the memory, not the CPU (see Francesc's class later this week)

timeit

- Precise timing of a function/expression
- Test different versions of a small amount of code, often used in interactive Python shell

```
from timeit import Timer

# execute 1 million times, return elapsed time(sec)
Timer("module.function(arg1, arg2)", "import module").timeit()

# more detailed control of timing
t = Timer("module.function(arg1, arg2)", "import module")
# make three measurements of timing, repeat 2 million times
t.repeat(3, 2000000)
```

- In ipython, you can use the `%timeit` magic command


DEMO

cProfile

- standard Python module to profile an entire application (`profile` is an old, slow profiling module)
- Running the profiler from command line:

```
python -m cProfile myscript.py
```

  options

```
-o output_file
-s sort_mode (calls,cumulative,name,...)
```

- from interactive shell/code:

```
import cProfile
cProfile.run(expression[,"filename.profile"])
```

`cProfile`, analyzing profiling results

- From interactive shell/code:

```
import pstat
p = pstat.Stats("filename.profile")
p.sort_stats(sort_order)
p.print_stats()
```

- Simple graphical description with RunSnakeRun


DEMO

Three more useful tools

- `pydoc`: creating documentation from your docstrings
  `pydoc [-w] module_name`
- `pylint`: check that your code respects standards

doctests

- `doctest` is a module that recognizes Python code in documentation and tests it
  - docstrings, rst or plain text documents
  - make sure that the documentation is up-to-date
- From command line:

```
python -m doctest -v example.py
```

- In a script:

```
import doctest
doctest.testfile("example.txt") # test examples in a file
doctest.testmod([module])     # test docstrings in module
```

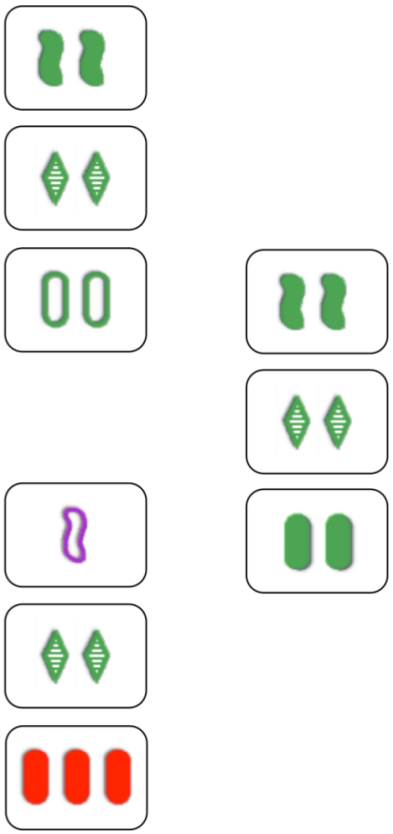

DEMO

Final thoughts

- Starting point: scientific code has slightly different needs than "regular" code, including the need to ensure correctness
- Agile programming methods, and testing in particular, go a long way toward this goal
- Agile programming in Python is as easy as it gets, there are no excuses not to do it!

The End

- Exercises after the break...



		1						
		2		3				4
			5			6		7
5			1	4				
	7						2	
				7	8			9
8		7			9			
4				6		3		
						5		